The KB paradigm and its application to interactive configuration.

Pieter Van Hertum, Ingmar Dasseville, Gerda Janssens, Marc Denecker

Department of Computer Science
KU LEUVEN
Leuven, BELGIUM
first.lastname@cs.kuleuven.be

submitted February 14, 2016; revised March 21, 2016; accepted May 2, 2016

Abstract

The knowledge base paradigm aims to express domain knowledge in a rich formal language, and to use this domain knowledge as a knowledge base to solve various problems and tasks that arise in the domain by applying multiple forms of inference. As such, the paradigm applies a strict separation of concerns between information and problem solving. In this paper, we analyze the principles and feasibility of the knowledge base paradigm in the context of an important class of applications: interactive configuration problems. In interactive configuration problems, a configuration of interrelated objects under constraints is searched, where the system assists the user in reaching an intended configuration. It is widely recognized in industry that good software solutions for these problems are very difficult to develop. We investigate such problems from the perspective of the KB paradigm. We show that multiple functionalities in this domain can be achieved by applying different forms of logical inferences on a formal specification of the configuration domain. We report on a proof of concept of this approach in a real-life application with a banking company.

KEYWORDS: Interactive Configuration, Knowledge Base Paradigm, Inferences, Applications of Declarative Systems

1 Introduction

In this paper, we investigate the application of knowledge representation and reasoning (KRR) to the problem of *interactive configuration*. In the past decades enormous progress in many different areas of computational logic was obtained. This resulted in a complex landscape with many declarative paradigms, languages and communities. One issue that fragments computational logic more than anything else

This is an extended version of a paper presented at the international symposium on Practical Aspects of Declarative Languages (PADL 2016), invited as a rapid communication in TPLP. The authors acknowledge the assistance of the conference program chairs Marco Gavanelli and John Reppy. This research was supported by the project GOA 13/010 Research Fund KU Leuven and projects G.0489.10, G.0357.12, and G.0922.13 of the Research Foundation - Flanders.

is the reasoning/inference task. Computational logic is divided in different declarative paradigms, each with its own syntactical style, terminology and conceptuology, and designated form of inference (e.g, deductive logic, logic programming, abductive logic programming, databases (query inference), answer set programming (answer set generation), constraint programming, etc.). Yet, in all of them declarative propositions need to be expressed. Take, e.g., "each lecture takes place at some time slot". This proposition could be an expression to be deduced from a formal specification if the task was a verification problem, or to be queried in a database, or it could be a constraint for a scheduling problem. It is, in the first place, just a piece of information and we see no reason why depending on the task to be solved, it should be expressed in a different formalism (classical logic, SQL, ASP, MiniZinc, etc.).

The Knowledge Base (KB) paradigm (Denecker and Vennekens 2008) was proposed as an answer to this. The KB paradigm applies a strict separation of concerns to information and problem solving. A KB system allows information to be stored in a knowledge base, and provides a range of inference methods. With these inference methods various types of problems and tasks can be solved using the *same knowledge base*. As such the knowledge base is neither a program nor a description of a problem, it cannot be executed or run. It is nothing but information. However, this information can be used to solve multiple sorts of problems. Many declarative problem solving paradigms are mono-inferential: they are based on one form of inference. In comparison, the KB-paradigm is multi-inferential. We believe that this implements a more natural, pure view of what declarative logic is aimed to be. The FO(·) KB project (Denecker and Vennekens 2008) is a research project that runs now for a number of years. Its aim is to integrate different useful language constructs and forms of inference from different declarative paradigms in one rich declarative language and a KB system. So far, it has led to the KB language FO(·) (Denecker and Ternovska 2008) and the KB system IDP (De Cat et al. 2016) which were used in the configuration experiment described in this paper.

An interactive configuration (IC) problem (McDermott 1982; Mittal and Frayman 1989; Fleischanderl et al. 1998; Junker and Mailharro 2003; Hadzic 2004) is an interactive version of a constraint solving problem. One or more users search for a configuration of objects and relations between them that satisfies a set of constraints. Industry abounds with interactive configuration problems: configuring composite physical systems such as cars and computers, insurances, loans, schedules involving human interaction, webshops (where clients choose composite objects), etc. However, building such software is renowned in industry as difficult and no broadly accepted solution methods are available (Felfernig et al. 2014; Axling and Haridi 1996). Building software support using standard imperative programming is often a nightmare (Barker and O'Connor 1989; Piller et al. 2014), due to the fact that (1) many functionalities need to be provided, (2) they are complex to implement, and (3) constraints on the configuration tend to get duplicated and spread out over the application, in the form of snippets of code performing various computations relative to the constraint (e.g., context dependent checks or propagations) which often leads to an unacceptable maintenance cost. This makes interactive con-

figuration an excellent domain to illustrate the advantages of declarative methods over standard imperative or object-oriented programming.

Our research question is: can we express the constraints of correct configurations in a declarative logic and provide the required functionalities by applying inference on this domain knowledge? This is a KRR question albeit a difficult one. In the first place, some of the domain knowledge may be complex. For an example in the context of a computer configuration problem, take the following constraint: *the total memory usage of different software processes that needs to be in main memory simultaneously, may not exceed the available RAM memory.* It takes an expressive knowledge representation language with aggregates to (compactly and naturally) express such a constraint. Many interactive configuration problems include complex constraints: various sorts of quantification, aggregates, definitions (sometimes inductive), etc. Moreover, an interactive configuration system needs to provide many functionalities: checking the validity of a fully specified configuration, correct and safe reasoning on a partially specified configuration (this involves reasoning on incomplete knowledge, sometimes with infinite or unknown domains), computing impossible values or forced values for attributes, generating sensible questions to the user, providing explanation why certain values are impossible, backtracking if the user regrets some choices, supporting the user by filling in his don't-cares while potentially taking into account a cost function, etc.

That declarative methods are particularly suitable for solving this type of problem has been acknowledged before, and several systems and languages have been developed (Hadzic 2004; Schneeweiss and Hofstedt 2011; Tiihonen et al. 2013; Vlaeminck et al. 2009). A first contribution of this paper is the analysis of IC problems from a Knowledge Representation point of view. We show that multiple functionalities in this domain can be achieved by applying different forms of logical inference on *the same* formal specification of the configuration domain. We define various sorts of inference and analyse them in terms of which different functionalities can be supplied. The second contribution is the reverse: we study the feasibility and usefulness of the KB paradigm in this important class of applications. The logic used in this experiment is the logic FO($\cdot$) (Denecker and Ternovska 2008), an extension of first-order logic (FO), and the system is the IDP system (De Cat et al. 2016). We discuss the complexity of (the decision problems of) the inference problems and why they are solvable, despite the high expressivity of the language and the complexity of inference. This research has its origin in an experimental IC system we developed in collaboration with industry. We evaluated our approach using the evaluation criteria of the knowledge-based configuration research (Felfernig et al. 2014). We conclude this paper with a discussion of related work in using knowledge-based systems for configuration and a comparison of our approach with these systems.

2 The FO(.) KB project

The language. FO($\cdot$) refers to the class of extensions of first order logic (FO) as is common in logic, e.g. FO(LFP) stands for the extension of FO with a least fixpoint construction (Immerman and Vardi 1997). Currently, the language of the

IDP system in the project is FO(T, ID, Agg, arit, PF) (Denecker and Ternovska 2008; Pelov et al. 2007): FO extended with types, definitions, aggregates, arithmetic and partial functions. Abusing notation, we will use FO($\cdot$) as an abbreviation for this language. Below, we introduce the aspects of the logic and its syntax on which this paper relies.

A specification. A *vocabulary* is a set Σ of type (denoted as Σ_T), predicate (denoted as Σ_P) and function symbols (denoted as Σ_F). Variables x, y, atoms A, FO-formulas φ are defined as usual. A predicate P of arity n has a type $[\tau_1, \ldots, \tau_n]$, a n-tuple of type symbols. A function of arity n has a type $[\tau_1, \ldots, \tau_n] \rightarrow \tau_{n+1}$, a $(n+1)$-tuple of type symbols. Aggregate terms are of the form $Agg(E)$, with Agg an aggregate function symbol and E an expression $\{(\overline{x}, F(\overline{x}))|\varphi(\overline{x})\}$, where φ is any FO-formula, F a function symbol and $\overline{x}$ a tuple of variables. Examples are the cardinality, sum, product, maximum and minimum aggregate functions. For example $sum\{(x, F(x))|\varphi(x)\}$ is read as $\Sigma_{x \in \{y|\varphi(y)\}} F(x)$. A *term* in FO($\cdot$) can be an aggregate term or a term as defined in FO. A *theory* is a set of FO($\cdot$) formulas.

A *partial set* on domain D is a function from D to $\{\mathbf{t}, \mathbf{u}, \mathbf{f}\}$. A partial set is two-valued (or total) if $\mathbf{u}$ does not belong to its range. A *(partial) structure* $\mathcal{S}$ consists of a domain D_τ for all types τ in Σ_T and an assignment of a partial set $\sigma^{\mathcal{S}}$ to each predicate or function symbol $\sigma \in (\Sigma_P \cup \Sigma_F)$, called the *interpretation* of σ in $\mathcal{S}$. The interpretation $P^{\mathcal{S}}$ of a predicate symbol P with type $[\tau_1, \ldots, \tau_n]$ in $\mathcal{S}$ is a partial set on domain $D_{\tau_1} \times \ldots \times D_{\tau_n}$. For a function F with type $[\tau_1, \ldots, \tau_n] \rightarrow \tau_{n+1}$, the interpretation $F^{\mathcal{S}}$ of F in $\mathcal{S}$ is a partial set on domain $D_{\tau_1} \times \ldots \times D_{\tau_n} \times D_{\tau_{n+1}}$. In case the interpretation of (a predicate or function symbol) σ in $\mathcal{S}$ is a two-valued set, we abuse notation and use $\sigma^{\mathcal{S}}$ as shorthand for $\{\overline{d}|\sigma^{\mathcal{S}}(\overline{d}) = \mathbf{t}\}$. The precision-order on the truth values is given by $\mathbf{u} <_p \mathbf{f}$ and $\mathbf{u} <_p \mathbf{t}$. It can be extended pointwise to partial sets and partial structures, denoted $\mathcal{S} \leq_p \mathcal{S}'$. Informally, this means that an interpretation has become more precise if tuples of domain elements that were previously mapped to unknown now map to true or false. Notice that total structures are the maximally precise ones. We will illustrate this precision relation in Example 2.1. We say that $\mathcal{S}'$ extends $\mathcal{S}$ if $\mathcal{S} \leq_p \mathcal{S}'$. We will sometimes use $\sigma^{\mathcal{S}}_x$ as shorthand for the set $\{\overline{d}|\overline{d} \in D_{\tau_1} \times \ldots \times D_{\tau_n} \wedge \sigma^{\mathcal{S}}(\overline{d}) = x\}$, with $x \in \{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$.

The associated theory $T_{\mathcal{S}}$ of a partial structure $\mathcal{S}$ is a representation of the information contained in $\mathcal{S}$ as a theory, which will be used in Section 4. It is defined by the following collection of constraints. For every predicate symbol P, this collection contains two sets of constraints:

$$\{P(\overline{d})|\overline{d} \in P^{\mathcal{S}}_{\mathbf{t}}\}$$
$$\{\neg P(\overline{d})|\overline{d} \in P^{\mathcal{S}}_{\mathbf{f}}\}$$

and two sets of constraints for every function symbol F:

$$\{F(\overline{d}) = e|(\overline{d}, e) \in F^{\mathcal{S}}_{\mathbf{t}}\}$$
$$\{\neg F(\overline{d}) = e|(\overline{d}, e) \in F^{\mathcal{S}}_{\mathbf{f}}\}$$

Given a partial structure $\mathcal{S}$, the domain structure $\mathcal{S}_D$ is the structure containing

only the domains of $\mathcal{S}$. It is easy to see that $\mathcal{S}$ contains the same information as $T_{\mathcal{S}} \cup \mathcal{S}_D$. A total structure[1] S is called *functionally consistent* if for each function F with type $[\tau_1, \ldots, \tau_n] \to \tau_{n+1}$, the interpretation F^S is the graph of a function $D_{\tau_1} \times \ldots \times D_{\tau_n} \mapsto D_{\tau_{n+1}}$. A partial structure $\mathcal{S}$ is functionally consistent if it has a functionally consistent two-valued extension. Unless stated otherwise, we will assume for the rest of this paper that all (partial) structures are functionally consistent.

A domain atom (domain term) is a tuple of a predicate symbol P (a function symbol F) and a tuple of domain elements $(d_1, \ldots, d_n)$. We will denote it as $P(d_1, \ldots, d_n)$ (respectively $F(d_1, \ldots, d_n)$). We say a domain term t of type τ is uninterpreted in $\mathcal{S}$ if $\{d | d \in D_\tau \wedge (t = d)^{\mathcal{S}} = \mathbf{u}\}$ is non-empty.

To define the satisfaction relation on theories, we extend the interpretation of symbols to arbitrary terms and formulas using the Kleene truth assignments (Kleene 1952). For a theory T and a partial structure $\mathcal{S}$, we say that $\mathcal{S}$ is a model of T (or in symbols $\mathcal{S} \vDash T$) if $T^{\mathcal{S}} = \mathbf{t}$ and $\mathcal{S}$ is two-valued. We sometimes abuse notation and write $T \vDash \varphi$ for the entailment relation, as a shorthand for "For every structure S such that $S \vDash T$, we have $S \vDash \varphi$.".

Example 2.1. To illustrate some of the concepts introduced above, assume a situation where we have some knowledge about printers, that have some type of connection. A vocabulary to model such knowledge can look as follows:

$$
\begin{aligned}
&\Sigma = \{ \\
&\quad \Sigma_T = \{printer, connection\} \\
&\quad \Sigma_P = \{PrinterConnection(printer, connection)\} \\
&\quad \Sigma_F = \{\} \\
&\}
\end{aligned}
$$

A structure $\mathcal{S}_0$ in which we have 2 printers P_1 and P_2 and 2 possible connections: USB and LAN, where we have no additional information, looks like:

$$
\begin{aligned}
&\mathcal{S}_0 = \{ \\
&\quad printer = \{P_1, P_2\} \\
&\quad connection = \{USB, LAN\} \\
&\quad PrinterConnection = \{(P_1, USB) \to \mathbf{u}, (P_2, USB) \to \mathbf{u}, \\
&\qquad (P_1, LAN) \to \mathbf{u}, (P_2, LAN) \to \mathbf{u}\} \\
&\}
\end{aligned}
$$

A more precise structure $\mathcal{S}_1 \geq_p \mathcal{S}_0$, containing the partial information that P_1 has

[1] Note the difference in typography between a partial structure $\mathcal{S}$ and a total structure S.

USB and P_2 certainly has no LAN connection looks like:

$$\begin{aligned}
&\mathcal{S}_1 = \{ \\
&\quad printer = \{P_1, P_2\} \\
&\quad connection = \{USB, LAN\} \\
&\quad PrinterConnection = \{(P_1, USB) \to \mathbf{t}, (P_2, USB) \to \mathbf{u}, \\
&\qquad (P_1, LAN) \to \mathbf{u}, (P_2, LAN) \to \mathbf{f}\} \\
&\}
\end{aligned}$$

A total structure $S_2 \geq_p \mathcal{S}_1$ containing full information can look like:

$$\begin{aligned}
&S_2 = \{ \\
&\quad printer = \{P_1, P_2\} \\
&\quad connection = \{USB, LAN\} \\
&\quad PrinterConnection = \{(P_1, USB) \to \mathbf{t}, (P_2, USB) \to \mathbf{t}, \\
&\qquad (P_1, LAN) \to \mathbf{f}, (P_2, LAN) \to \mathbf{f}\} \\
&\}
\end{aligned}$$

Inference tasks. In the KB paradigm, a specification is a bag of information. This information can be used for solving various problems by applying a suitable form of inference on it.

FO is standardly associated with deduction inference: a deductive inference task takes as input a pair of theory T and sentence φ, and returns $\mathbf{t}$ if $T \models \varphi$ and $\mathbf{f}$ otherwise. This is well-known to be undecidable for FO, and by extension for FO$(\cdot)$. However, to provide the required functionality of an interactive configuration system we can use simpler forms of inference. Indeed, in many such domains a fixed finite domain is associated with each unknown configuration parameter.

A natural format in logic to describe these finite domains is by a partial structure with a finite domain. Also other data that are often available in such problems can be represented in that structure. As such various inference tasks are solvable by finite domain reasoning and become decidable. Below, we give the base forms of inference for our KB system and recall their complexity when using finite domain reasoning. We assume a fixed vocabulary Σ and theory T and query. Our complexities are given in function of the domain size.

Modelexpand$(T, \mathcal{S})$: input: theory T and partial structure $\mathcal{S}$; output: a model I of T such that $\mathcal{S} \leq_p I$ or $UNSAT$ if there is no such I. Modelexpand (Wittocx et al. 2008) is a generalization for FO$(\cdot)$ theories of the modelexpansion task as defined in Mitchell et al. (Mitchell and Ternovska 2005). Complexity of deciding the existence of a modelexpansion is in **NP**. Structure S_2 in Example 2.1 is the output of Modelexpand$(T, \mathcal{S}_1)$, with $\mathcal{S}_1$ as in Example 2.1, and T a theory consisting of the constraint that every printer has exactly one connection.

Modelcheck(T, S): input: a total structure S and theory T over the vocabulary interpreted by S; output is the boolean value $S \models T$. Note that Modelcheck

is a degenerate case of the Modelexpand inference, with $\mathcal{S}$ a total structure. Complexity is in **P**.

Minimize$(T, \mathcal{S}, t)$**:** input: a theory T, a partial structure $\mathcal{S}$ and a term t of numerical type; output: a model $I \geq_p \mathcal{S}$ of T such that the value t^I of t is minimal. The term t represents a numerical expression whose value has to be minimized. This is an extension to the modelexpand inference. The complexity of deciding that a certain t^I is minimal, is in $\mathbf{\Delta_2^P}$.

Propagate$(T, \mathcal{S})$**:** input: theory T and partial structure $\mathcal{S}$; output: the most precise partial structure $\mathcal{S}_r$ such that for every model $I \geq_p \mathcal{S}$ of T it is true that $I \geq_p \mathcal{S}_r$. The complexity of deciding that a partial structure $\mathcal{S}'$ is $\mathcal{S}_r$ is in $\mathbf{\Delta_2^P}$. Note that we assume that all partial structures are functionally consistent, which implies that we also propagate functional integrity constraints.

Query$(\mathcal{S}, E)$**:** input: a (partial) structure $\mathcal{S}$ and a set expression $E = \{\overline{x} \mid \varphi(\overline{x})\}$; output: the set $A_Q = \{\overline{x} \mid \varphi(\overline{x})^{\mathcal{S}} = \mathbf{t}\}$. Complexity of deciding that a set A is A_Q is in **P**.

Approximative versions exist for some of these inferences, with lower complexity (Vlaeminck et al. 2009). More inferences exist, such as simulation of temporal theories in FO$(\cdot)$ (Bogaerts et al. 2014), but were not used in the experiment.

3 Interactive Configuration

In an IC problem, one or more users search for a configuration of objects and relations between them that satisfies a set of constraints.

Typically, the user is not aware of all constraints. There may be too many of them to keep track of. Even if the human user can oversee all constraints that he needs to satisfy, he is not a perfect reasoner and cannot comprehend all consequences of his choices. This in its own right makes such problems hard to solve. The problems get worse if the user does not know about the relevant objects and relations or the constraints on them, or if the class of involved objects and relations is large, if the constraints get more complex and more "irregular" (e.g., exceptions), if more users are involved, etc. On top of that, the underlying constraints in such problems tend to evolve quickly. All these complexities occur frequently, making the problem difficult for a human user. In such cases, computer assistance is needed: the human user chooses and the system assists by guiding him through the search space.

For a given IC problem, an IC system has information on that problem. There are a number of stringent rules to which a configuration should conform, and besides this there is a set of parameters. Parameters are the open fields in the configuration that need to be filled in by the user or decided by the system.

3.1 Running example: Domain knowledge

A simplified version of the application in Section 5.1 is used in Section 4 as running example. We introduce the domain knowledge of this example here.

Example 3.1. Software on a computer has to be configured for different employees.

Table 1. Example data

PriceOf		**PreReq**		**MaxCost**		**IsOS**
software	*int*	*software*	*software*	*employee*	*int*	*software*
Windows	60	Office	Windows	Secretary	100	Windows
Linux	20	LaTeX	Linux	Manager	150	Linux
LaTeX	10					
Office	30					
DualBoot	40					

Table 1 contains the information on the software, the requirements, the budgets of the employees and the prices of software. Available software is Windows, Linux, LaTeX, Office and a DualBoot system. Each software item has a price, which can be seen in column **PriceOf**. Column **PreReq** specifies which software is required for other software. Every type of employee has a budget, provided in column **MaxCost**. **IsOs** lists the pieces of software that are operating systems. Next to the information in the table, we know that if more than one OS is installed, a DualBoot System is required.

3.2 Subtasks of an interactive configuration system

Any system assisting a user in interactive configuration must be able to perform a set of subtasks. We look at important subtasks that an interactive configuration system should support.

Subtask 1: Acquiring information from the user

The first task of an IC system is acquiring information from the user. The system needs to get a value for a number of parameters of the configuration from the user. There are several options: the system can ask questions to the user, it can make the user fill in a form containing open text fields, dropdown-menus, checkboxes, etc. Desirable aspects would be to give the user the possibility to choose the order in which he gives values for parameters and to omit filling in certain parameters (because he does not know or does not care). For example, in the running example a user might need a LaTeX-package, but he does not care about which OS he uses. In that case the system will decide in his place that a Linux system is required. Since a user is not fully aware of all constraints, it is possible that he inputs conflicting information. This needs to be handled or avoided.

Subtask 2: Generating consistent values for a parameter

After a parameter is selected (by the user or the system) for which a value is needed, the system can assist the user in choosing these values. A possibility is that the system presents the user with a list of all possible values, given the values for other parameters and the constraints of the configuration problem. Limiting the user with this list makes that the user is unable to input inconsistent information.

Subtask 3: Propagation of information

Assisting the user in choosing values for the parameters, a system can use the constraints to propagate the information that the user has communicated. This can be used in several ways. A system can communicate propagations through a GUI, for example by coloring certain fields red or graying out certain checkboxes. Another way is to give a user the possibility to explicitly ask "what if"-questions to the system. In Example 3.1, a user can ask the system what the consequences are if he was a secretary choosing an Office installation. The system answers that in this case a Windows installation is required, which results in a Linux installation becoming impossible (due to budget constraints) and as a consequence it also derives the impossibility of installing LaTeX.

Subtask 4: Checking the consistency for a value

When it is not possible/desirable to provide a list of possible values, the system checks that the value the user has provided is consistent with the known data and the constraints.

Subtask 5: Checking a configuration

If a user makes manual changes to a configuration, the system provides him with the ability to check if his updated version of the configuration still conforms to all constraints.

Subtask 6: Autocompletion

If a user has finished communicating all his preferences, the system autocompletes the partial configuration to a full configuration. This can be done arbitrarily (a value for each parameter such that the constraints are satisfied) or the user can have some other parameters like total cost, that have to be optimized.

Subtask 7: Explanation

If a supplied value for a parameter is not consistent with other parameters, the system can explain this inconsistency to the user. This can be done by showing minimal sets of parameters with their values that are inconsistent, by showing (visualizations of) constraints that are violated or by combinations of both. It can also explain to the user why certain automatic choices are made, or why certain choices are impossible.

Subtask 8: Backtracking

It is not unthinkable that a user makes a mistake, or changes his mind after seeing consequences of choices he made. Backtracking is an important subtask for a configuration system, and can be supported in numerous ways. The simplest way is

a simple back button, which reverts the last choice a user made. A more involved option is a system where a user can select any parameter and erase his value for that parameter. The user can then decide this parameter at a later timepoint. Even more complex is a system where a user can supply a value for a parameter and if it is not consistent with other parameters the system shows him which parameters are in conflict and proposes other values for these parameters such that consistency can be maintained.

4 Interactive Configuration in the KB paradigm

To analyze the IC problem from the KB point of view, we aim at formalizing the subtasks of Section 3 as inferences. In this paper we do not deal with user interface aspects. For a given application, our knowledge base consists of a vocabulary Σ, a theory T expressing the configuration constraints and a partial structure $\mathcal{S}$. Initially, $\mathcal{S}_0$ is the partial structure that contains the domains of the types and the input data. During IC, $\mathcal{S}_0$ will become more and more precise partial structures $\mathcal{S}_i$ due to choices made by the user. For IC, the KB also contains $L_{\mathcal{S}_0}$, the set of all uninterpreted domain atoms/terms[2] in $\mathcal{S}_0$. These domain terms are the logical formalization of the parameters of the IC problem. Σ and T are fixed. As will be shown in this section, all subtasks can be formalized by (a combination of) inferences on this knowledge base consisting of $\Sigma, T, \mathcal{S}_0, L_{\mathcal{S}_0}$ and information gathered from the user.

Example 4.1. Continuing Example 3.1, use vocabulary Σ:

$$
\begin{aligned}
\Sigma = \\
&\Sigma_T = \{software,\ employee,\ int\} \\
&\Sigma_P = \{Install(software),\ IsOS(software),\ PreReq(software, software)\} \\
&\Sigma_F = \{PriceOf(software) : int,\ MaxCost(employee) : int, \\
&\qquad Cost : int,\ Requester : employee\}
\end{aligned}
$$

The initial partial structure $\mathcal{S}_0$ consists of:

$$
\begin{aligned}
&employee \rightarrow \{Secretary, Manager\} \\
&software \rightarrow \{Windows, Linux, LaTeX, Office, DualBoot\}
\end{aligned}
$$

and interpretations for *MaxCost (employee):int, IsOs(software), PriceOf(software): int* and *PreReq(software, software)* as can be seen in Table 1. All symbols from Σ that are not specified above are assumed to be fully unknown in $\mathcal{S}_0$.
The set of parameters $L_{\mathcal{S}_0}$ is:

$$
\begin{aligned}
&\{Requester, Install(Windows), Install(Linux), \\
&\ Install(Office), Install(LaTeX), Install(DualBoot), Cost\}
\end{aligned}
$$

[2] In the rest of this paper, a domain atom is treated as a term that evaluates to true or false.

The theory T consists of the following constraints:

$\forall s1\, s2 :\ Install(s1) \wedge PreReq(s1, s2) \Rightarrow Install(s2).$
// The total cost is the sum of the prices of all installed software.
$Cost = sum\{(s, PriceOf(s))|Install(s)\}.$
$Cost \leq MaxCost(Requester).$
$\exists s :\ Install(s) \wedge IsOS(s).$
$Install(Windows) \wedge Install(Linux) \Rightarrow Install(DualBoot).$

Subtask 1: Acquiring information from the user

Key in IC is collecting information from the user on the parameters. During the run of the system, the set of parameters that are still open changes. In our KB system a derived inference (a combination of the inferences as introduced in Section 2) is used to calculate this set of parameters. Complexity results of derived inferences stem from basic results formulated by Mitchell and Ternovska (2005) and the observation that modelchecking is polynomial in the size of the domain.

Definition 4.2. Calculating uninterpreted terms.
GetOpenTerms$(T, \mathcal{S})$ is the derived inference with input a theory T, a partial structure $\mathcal{S} \geq_p \mathcal{S}_0$ and the set $L_{\mathcal{S}_0}$ of terms. Output is a set of terms such that for every term t in that set, there exist models I_1 and I_2 of T that extend $\mathcal{S}$ ($I_1, I_2 \geq_p \mathcal{S}$) for which $t^{I_1} \neq t^{I_2}$. Or formally:

$$\{l|l \in L_{\mathcal{S}_0} \wedge \{d|(l = d)^{\mathcal{S}'} = \mathbf{u}\} \neq \emptyset \wedge \mathcal{S}' = Propagate(T, \mathcal{S})\}$$

The complexity of deciding whether a given set of terms A is the set of uninterpreted terms is in $\mathbf{\Delta_2^P}$.

An IC system can use this set of terms in a number of ways. It can use a metric to select a specific term, which it can pose as a direct question to the user. It can also present a whole list of these terms at once and let the user pick one to supply a value for. In Section 5.1, we discuss two different approaches we implemented for this project.

Example 4.3. In Example 4.1, the parameters and domains are already given. Assume that the user has chosen the value *Manager* for *Requester*, true for *Install(Windows)* and false for *Install(Linux)*. The system will return *GetOpenTerms*$(T, \mathcal{S})$ = {*Install(Office), Install(DualBoot), Cost*}.

Subtask 2: Generating consistent values for a parameter

A domain element d is a possible value for term t if there is a model $I \geq_p \mathcal{S}$ such that $(t = d)^I = \mathbf{t}$.

Definition 4.4. Calculating consistent values.
GetConsistentValues$(T, \mathcal{S}, t)$ is the derived inference with input a theory T, a partial structure $\mathcal{S}$ and a term $t \in GetOpenTerms(T, \mathcal{S})$. Output is the set

$$\{t^I|\ I \text{ is a model of } T \text{ extending } \mathcal{S}\}$$

The complexity of deciding that a set P is the set of consistent values for t is in $\mathbf{\Delta_2^P}$.

Example 4.5. The consistent values for *Requester* given T and the initial partial structure $\mathcal{S}_0$ from Example 4.1 is:

$$GetConsistentValues(T, \mathcal{S}, Requester) = \{Secretary, Manager\}$$

Consistent values for other terms are the integers for *Cost* and {*true, false*} for the others.

Subtask 3: Propagation of information

It is informative for the user that he can see the consequences of assigning a particular value to a parameter.

Definition 4.6. Calculating Consequences.
PosConsequences$(T, \mathcal{S}, t, a)$ and **NegConsequences**$(T, \mathcal{S}, t, a)$ are derived inferences with input a theory T, a partial structure $\mathcal{S}$, an uninterpreted term $t \in$ GetOpenTerms$(T, \mathcal{S})$ and a domain element $a \in$ GetConsistentValues$(T, \mathcal{S}, t)$. As output it has a set C^+, respectively C^- of tuples (q, b) of uninterpreted terms and domain elements. $(q, b) \in C^+$, respectively C^- means that the choice a for t entails that q will be forced, respectively prohibited to be b. Formally,

$$\begin{aligned} C^+ = \{(q,b) \mid\ & (q=b)^{\mathcal{S}'} = \mathbf{t} \wedge (q=b)^{\mathcal{S}} = \mathbf{u} \\ & \wedge\ \mathcal{S}' = Propagate(T, \mathcal{S} \cup \{t=a\}) \\ & \wedge\ q \in GetOpenTerms(T, \mathcal{S}) \setminus \{t\}\ \} \\ C^- = \{(q,c) \mid\ & (q=c)^{\mathcal{S}'} = \mathbf{f} \wedge (q=c)^{\mathcal{S}} = \mathbf{u} \\ & \wedge\ \mathcal{S}' = Propagate(T, \mathcal{S} \cup \{t=a\}) \\ & \wedge\ q \in GetOpenTerms(T, \mathcal{S}) \setminus \{t\}\ \} \end{aligned}$$

The complexity of deciding whether a set P is C^+ or C^- is in $\mathbf{\Delta_2^P}$.

Example 4.7. Say the user has chosen $Requester = Secretary$ and wants to know the consequences of making $Install(Windows)$ true. The output in this case contains $(Install(LaTeX), \mathbf{f})$ in $PosConsequences(T, \mathcal{S}, t, a)$ and $(Install(LaTeX), \mathbf{t})$ in $NegConsequences(T, \mathcal{S}, t, a)$ since this combination is too expensive for a secretary. Note that there is not always such a correspondence between the positive and negative consequences. For example, when deriving a negative consequence for *Cost*, this does not necessarily imply a positive consequence.

Subtask 4: Checking the consistency for a value

A value d for a term t is consistent if there exists a model of T in which $t = d$ that extends the partial structure representing the current state.

Definition 4.8. Consistency Checking.
CheckConsistency$(T, \mathcal{S}, t, d)$ is the derived inference with input a theory T, a partial structure $\mathcal{S}$, an uninterpreted term t and a domain element d. Output is a boolean b that represents whether $\mathcal{S}$ extended with $t = d$ still satisfies T. Formally we return **t** if

$$(\mathcal{S} \cup \{t^{\mathcal{S}} = d\}) \models T$$

and **f** otherwise. Complexity of deciding if a value d is consistent for a term t is in **NP**.

Example 4.9. If a user has chosen $Install(Windows)$ and $Install(LaTeX)$ to be true, then *Manager* will and *Secretary* will not be a consistent answer for *Requester*.

Subtask 5: Checking a configuration

Once the user has constructed a 2-valued structure S and makes manual changes to it, he may need to check if all constraints are still satisfied. A theory T is checked on a total structure S by calling $Modelcheck(T, S)$, with complexity in **P**.

Subtask 6: Autocompletion

If a user is ready communicating his preferences (Subtask 1) and there are undecided terms left which he does not know or care about, the user may want to get a full configuration (i.e. a total structure). This is computed by modelexpand. In particular:

$$I = Modelexpand(T, \mathcal{S})$$

In many of those situations the user wants to have a total structure with, for example, a minimal cost (given some term representing the cost t). This is computed by minimize:

$$I = Minimize(T, \mathcal{S}, t)$$

Example 4.10. Assume the user is a secretary and all he knows is that he needs Office. He chooses *Secretary* for *Requester* and true for $Install(Office)$ and calls autocompletion. A possible output is a structure S where for the remaining parameters, a choice is made that satisfies all constraints, e.g., $Install(Windows)^S = \mathbf{t}$, $Install(DualBoot)^S = \mathbf{t}$ and the other $Install$ atoms false. This is not a cheapest solution (lowest cost). By calling minimize using cost-term *Cost*, the DualBoot is dropped.

Subtask 7: Explanation

It is clear that a whole variety of options can be developed to provide different kinds of explanations to a user. If a user supplies an inconsistent value for a parameter, options can range from calculating an inconsistent subset of the theory T (1) to giving a proof of inconsistency as in (Pontelli and Son 2006) (2), to calculating

a partial subconfiguration that has this inconsistency (3). UnsatSubstructure is a logical inference for option 3.

Definition 4.11. Calculating inconsistent structures.
UnsatSubstructure$(T, \mathcal{S})$ is a derived inference with input a theory T and a partial structure $\mathcal{S}$ that cannot be extended to a model of T and as output all (partial) structures $\mathcal{S}' \leq_p \mathcal{S}$ such that $\mathcal{S}'$ cannot be extended to a model I of T. Formally, we return:

$$\{\mathcal{S}' | \mathcal{S}' \leq_p \mathcal{S} \wedge \neg(\exists I \geq_p \mathcal{S}' \wedge I \vDash T)\}$$

Complexity of deciding if a set is an inconsistent substructure is in $\mathbf{co-NP}$.

The inference in Definition 4.12 calculates an inconsistent subtheory.

Definition 4.12. Calculating inconsistent theories.
UnsatSubtheory$(T, \mathcal{S})$ is a derived inference with input theory T and a partial structure $\mathcal{S}$ such that there does not exist a model I, extending S, satisfying T. The inference has as output all theories T' such that $T' \subseteq T$ and there is no model satisfying T, extending $\mathcal{S}$. Formally, we return:

$$\{T' | T' \subseteq T \wedge \neg(\exists I \geq_p \mathcal{S} \wedge I \vDash T')\}$$

Complexity of deciding if a theory is such an inconsistent theory is in $\mathbf{co-NP}$.

Note that Definition 4.11 and 4.12 do not make any statements of minimality.

Using the associated theory $T_{\mathcal{S}}$ and domains structure $\mathcal{S}_D$ of a partial structure $\mathcal{S}$, it is possible to consider calculating minimally precise partial configurations as a special case of calculating a minimal inconsistent subset of the theory. As in (Shchekotykhin et al. 2014), we can introduce a "background theory" $B \subset T \cup T_S$ (a subset of the theory in which there are assumed to be no conflicts). We define multiple derived logical inferences, with different degrees of minimality (not-minimal, subset-minimal and minimal in size) of increasing complexity, able to provide explanations to the user.

Definition 4.13. Calculating inconsistent theories with a background.
UnsatSubtheory$(T, \mathcal{S}, B)$ is a derived inference with input theory T, a partial structure $\mathcal{S}$ and a background theory $B \subseteq T \cup T_{\mathcal{S}}$ such that there does not exist a model I, with the domains as in $\mathcal{S}_D$ satisfying $T \cup T_{\mathcal{S}}$ (or equivalently: extending $\mathcal{S}$ and satisfying T), but there is a model satisfying B. The inference has as output all theories T' such that $B \subseteq T' \subseteq T \cup T_{\mathcal{S}}$ and there is no model satisfying T. Formally, we return:

$$\{T' | B \subseteq T' \subseteq (T \cup T_S) \wedge \neg(\exists I \geq_p \mathcal{S}_D \wedge I \vDash T')\}$$

Complexity of deciding if a theory is such an inconsistent theory is in $\mathbf{co-NP}$.

Definition 4.14. Calculating minimal inconsistent theories with a background.
MinimalUnsatTheory$(T, \mathcal{S}, B)$ is a derived inference with input theory T, a partial structure $\mathcal{S}$ and a background theory B as above. Output is the subset of subset

minimal theories from *UnsatSubtheory(*$T, \mathcal{S}, B$*)*. Complexity of deciding if a set is a subset minimal inconsistent theory is in Δ_2^P.

Definition 4.15. Calculating minimum inconsistent theories with a background.
MinimumUnsatTheory($T, \mathcal{S}, B$**)** is a derived inference with input theory T, a partial structure $\mathcal{S}$ and a background theory B as above. Output is the subset of cardinality minimal theories from *MinimalUnsatTheory(*$T, \mathcal{S}, B$*)*. Complexity of deciding if a set is a cardinality minimal inconsistent theory is Π_2^P.

Note that Definition 4.11 is equivalent to calculating a minimal inconsistent subset of a theory $T \cup T_S$, with $B = T$, if you translate the output back to a pair of a theory and a structure. Definition 4.12 is equivalent to calculating a minimal inconsistent subset of a theory $T \cup T_{\mathcal{S}}$, with $B = T_{\mathcal{S}}$, if you translate the output back to a pair of a theory and a structure.

In recent literature multiple approaches are discussed, all mapping to one of our explanation-related inferences. QuickXPlain (Junker 2004) is an algorithm that implements Definition 4.13. The Hitting Set Directed Acyclic Graph (HSDAG) (Reiter 1987) algorithm calculates subset minimal inconsistent theories (Definition 4.14, as in different ASP solvers (Shlyakhter et al. 2003; Syrjänen 2006). Implementations of Definition 4.15 have been described in (Lynce and Silva 2004) and (Zhang et al. 2006). In our experiment, we have an implementation of Definition 4.14 (Wittocx et al. 2009), where we do however do not calculate the entire set of subset minimal theories. We only calculate one, which gives one explanation of the inconsistency. If the user resolves that problem, he can ask for a new explanation which will point to another reason of inconsistency. This process is reiterated untill all problems are resolved.

Example 4.16. We show a minimal inconsistent subtheory in a situation with T as in Example 4.1 and $\mathcal{S}_i$, a partial structure representing an intermediate configuration where a user started with $\mathcal{S}_0$ and has chosen *Secretary* for *Requester*, and wants to Install *Office* and *Linux*. This is not possible, and as such, the user asks the system for an explanation in the form of a minimal inconsistent theory. A possible minimal inconsistent theory with $B = \emptyset$, is:

$$(Install(Office) \wedge PreReq(Office, Windows)) \Rightarrow Install(Windows).$$
$$Cost = sum\{(s, PriceOf(s))|Install(s)\}.$$
$$Cost \leq MaxCost(Requester).$$

This means that there is no valid configuration because Windows needs to be installed as prerequisite for Office, and the total cost then exceeds the budget of a Secretary.

Subtask 8: Backtracking

If a value for a parameter is not consistent, the user has to choose a new value for this parameter, or backtrack to revise a value for another parameter. In Section 3.2

we discussed three options of increasing complexity for implementing backtracking functionality. Erasing a value for a parameter is easy to provide in our KB system, and since this is a generalization of a back button (erasing the last value) we have a formalization of the first two options. Erasing a value d for parameter t in a partial structure $\mathcal{S}$ is simply modifying $\mathcal{S}$ such that $(t = d)^{\mathcal{S}} = \mathbf{u}$. As with explanation, a number of more complex options can be developed. We look at one possibility. Given a partial configuration $\mathcal{S}$, a parameter p and a value d that is inconsistent for that parameter, calculate a minimal set of previous choices that need to be undone such that this value is possible for this parameter. The converse of this problem is very well known under the name of maximum satisfiability problems. In other words, you want to hold on to as much of the structure as possible while ensuring satisfiability.

This problem is closely related to the explanation subtask (Heras et al. 2011; Marques-Silva and Planes 2008). You can imagine the explanation problem as asking the system to point out a mistake in your reasoning. However, solving this mistake will not guarantee you have not made any other mistake in the rest of the problem. What we actually need is a minimal set of things we can remove, so every problem is solved simultaneously.

So more formally, we can use Definition 4.11 and calculate $UnsatStructure(T \wedge (t = d), \mathcal{S})$. This inference calculates a set A of sets of previous choices that together are inconsistent. Undoing an arbitrary choice in all of these sets results in a partial subconfiguration $\mathcal{S}'$ of $\mathcal{S}$ such that d is a possible value for t in $\mathcal{S}'$. To find the maximal partial subconfiguration $\mathcal{S}'$ that satisfies that property, the minimal hitting set (Reiter 1987) of all sets in A has to be calculated.

5 Proof of Concept

5.1 Implementation

In this section we will describe the developed application and implementation. Our application has a simple client-server architecture. The server plays the role of the reasoning engine, which is mainly a thin wrapper around the IDP system. The client consists of a GUI made in QML (QML 2015) as front-end.

The server converts IDP into a stateless server which is accessible through the web. The client application sends the necessary information, consisting of theories, partial structures and choices, to this server and the server executes the needed inferences. This is a design which involves repeatedly sending over the choices a user has made, but it allows for a very simple architecture to show the feasibility of our design.

This implementation was done in cooperation with Adaptive Planet, a consulting company (Adaptive Planet 2015) that developed the user interface, and an international banking company that provided us with a substantial configuration problem for testing purposes. More practical information about this implementation, some screenshots, a downloadable demo and another example of a configuration system

developed with IDP as a reasoning engine (a simpler course configuration demo) can be found at: `http://www.configuration.tk`.

5.1.1 The Reasoning Engine

As explained before, the application we developed was built on the knowledge base system IDP, which was not developed specifically with configuration problems in mind. It provides the basic inferences listed at the end of Section 2. The goal of this experiment was to check if this general infrastructure could be readily applied to applications such as configuration.

In Section 4 we showed how the tasks which are needed for configuration relate to the infrastructure provided by IDP. Our main implementation task was to convert these specifications to code. Some subtasks such as autocompletion did not require any extra work, as this functionality is directly available as the modelexpand inference. Some functionality, e.g. calculating consequences, did require some work but the existing functionality provided almost all needed components.

We mainly use the existing forms of inference that are readily available in the IDP system. No dedicated or specialized algorithms are used for the configuration subtasks. This proves the point that the KB-paradigm is very flexible but this also means that we had relatively little impact upon the efficiency of our server. However, the system ended up being quite responsive and we could conclude that IDP (and by extension the KB-paradigm) passed the test for usefulness in this application.

5.1.2 User Interface

Apart from a reasoning engine, it is also necessary to have an accessible front end so the user has easy access to the multitude of functionalities which are available. The front end consists of an application written in the Qt framework using QML (QML 2015) and connects to a configuration engine over the web. For the purposes of our demo, we developed two different graphical interfaces:

Wizard In the wizard interface, the user is interrogated and he answers on subsequent questions selected by the system, using the $GetOpenTerms$ inference. An important side note here is that the user can choose not to answer a specific question, for instance because he cannot decide as he is missing relevant information or because he is not interested in the actual value (at this point). These parameters can be filled in at a later timepoint by the user, or by the system, using propagation, or in case the user calls autocompletion.

Drill-Down In the drill-down interface, the user sees a list of the still open parameters, and can pick which one he wants to fill in next. This interface is useful if the user is a bit more knowledgeable about the specific configuration and wants to give the values in a specific order.

In both interfaces the user is assisted in the same way when he enters data. When

he or the system selects a parameter, he is provided with a dropdown list of the possible values, using the *GetConsistentValues* inference. Before committing to a choice, he is presented with the consequences of his choice, using the calculate consequences inference. The nature of the system guarantees a correct configuration and will automatically give the user support using all information it has (from the knowledge base, or received from the user).

5.2 Evaluation

5.2.1 Evaluation Criteria

When evaluating the quality of software (especially when evaluating declarative methods), scalability (data complexity) is often seen as the most important quality metric. Naturally when using an interactive configuration system, performance is important. However, in the configuration community it is known that reasoning about typical configuration problems is relatively easy and does not exhibit real exponential behavior (Tiihonen et al. 2013). Also, depending on the application, it is reasonable to expect the number of parameters to be limited, since humans need to fill in the configuration in the end. When developing a configuration system, challenges lie in the complexity of the knowledge, its high volatility and the complex functionalities to be built. To get a more complete view of the performance of a configuration system, we chose to evaluate on a larger set of different evaluation criteria. In recent literature (Felfernig et al. 2014) nine evaluation criteria are used to differentiate between different paradigms used for configuration. In Section 6, ten other approaches will be discussed and compared to our solution using the same nine criteria.

Grapical Modeling Concepts (C1) is supported if there are standard graphical modeling techniques available that visualize configuration knowledge. They improve understandability, development time and maintenance of new knowledge bases.

Component Oriented modeling (C2) is a criterion that states that the modeling language is a natural language that allows knowledge base design on the basis of real-world concepts: types, relations, hierarchies, etc.

Automated Consistency Maintenance (C3) can be broken down to two categories. Firstly, a system can have support for a priori automated consistency maintenance. This helps a developer write consistent constraints and verifying correctness while writing the knowledge base. Secondly, runtime automated consistency maintenance supports the end user, by guaranteeing that every intermediate configuration he can make, can be extended to a valid configuration.

Modularization concepts are available (C4) if the modeling language is modular and has support for adding additional structure to the knowledge base, for example by organizing the constraints in blocks or groups.

Maintainability (C5) relates to the adaptability of the knowledge base if the background information changes. This background information is volatile, it is for example depending on ever-changing company policies. As such, it is vital

that when that information changes, the system can be easily adapted. When using custom software, all tasks using domain knowledge (like rules and policies) need their own program code. The domain knowledge is scattered all over the program. If this policy changes, a programmer has to find all snippets of program code that are relevant for guarding this policy and modify them. This results in a system that is hard to maintain, hard to adapt and error-prone. Every time the domain knowledge changes, a whole development cycle has to be run through again. Some systems have support for intelligent knowledge base navigation tools for complex knowledge spaces.

Model-based (C6) means that a knowledge base in the system expresses exactly what it means for a configuration to be valid. This in contrast to rule-based configuration, where a knowledge base also contains problem solving knowledge (i.e. information on how the rules should be used/fired).

Efficiency (C7) relates to efficiency and scalability of the reasoning engine.

Ability to solve generative problem settings (C8) means that the language supports talking about component types instead of specific objects. A system supports generic constraints if it allows for constraints that apply to every instance of a component type on which the constraint is defined. For example, the first constraint of Theory T in Example 4.1 is a generic constraint about all software, without explicitly naming the individual pieces of software.

Ability to provide explanations (C9) means that the system is able to communicate reasons for inconsistencies or explain why certain choices are forced/prohibited.

5.2.2 Evaluation

The criteria discussed in previous section are a good way to evaluate the KB implementation of a configuration system. We evaluate our implementation and the IDP system with these criteria.

Grapical Modeling Concepts (C1). IDP has no support for graphical modeling of domain knowledge and we did not develop any tools for this experiment. However, it must be noted, that a highly expressive and readable modeling language often makes graphical modeling obsolete.

Component Oriented modeling (C2). The FO$(\cdot)$ language used in this experiment is an extension of typed first-order logic. First-order logic is about a small set of connectives: $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow, \exists, \forall$. These connectives are also the basic connectives of information used by humans. Classical logic is a good KR language because it has a very clear informal semantics. It does however not suffice for knowledge representation. FO$(\cdot)$ extends classical logic with a number of extensions that arise from research in AI and KR, such as aggregates, inductive definitions, types, ... This makes FO$(\cdot)$ a suited modeling language for a configuration system.

Automated Consistency Maintenance (C3). A priori consistency maintenance is supported in the implementation by using the explanation inferences. If the developer has a collection of constraints that is consistent, it is possible

to evaluate if a new constraint leads to an inconsistency and ask the system what other constraints it conflicts with, using for example definition 4.14. At runtime consistency maintenance is partially supported, by using the inferences in subtask 2, 3 and 4. These inferences are theoretically able to guarantee consistency, but due to computational limitations, approximate versions can be used. These are not always able to give the same guarantees.

Modularization concepts are available (C4). The implemented configuration system is modular, since a knowledge base can consist of multiple theories and structures, that together make up the specification. The explanation inference allows that a user selects background constraints, as in definition 4.14, and in this way he can choose about which constraints he needs feedback.

Maintainability (C5). The development of a KB system with a centrally maintained knowledge base makes the knowledge directly available, readable and adaptable. A well-known advantage of this approach is in maintainability: if domain information changes, the developer can easily modify the knowledge base. The current implementation does however have no additional support for knowledge base navigation tools.

Model-based (C6). The FO(·) modeling methodology is based on formulating the properties of a correct configuration in a natural way, such that the models of a specification correspond with configurations. This is inherently a model-based approach.

Efficiency (C7). As explained in Section 5.1, we have only written a thin layer upon existing software which did not target configuration problems specifically. The performance of the IDP system has been tested extensively in other contexts (Jansen et al. 2014; Bruynooghe et al. 2015). The reasoning engine for IDP is very similar in performance to mainstream ASP solvers (Calimeri et al. 2014). Their performance was tested more extensively in the context of configuration by Tiihonen et al. (2013). It is also very difficult to reliably compare the response times for interactive systems. Standard benchmarking techniques in software engineering traditionally use instances which need multiple minutes to solve. In this setting we aim for subsecond response times, for which no standard benchmarks are available as far as we are aware.

In this experiment (a configuration task with 300 parameters and 650 constraints), our users reported a response time of a half second on average with outliers up to 2 seconds. Note that the provided implementation was a naive prototype and optimizing the efficiency of the implemented algorithms is still possible in a number of ways.

Ability to solve generative problem settings (C8). FO(·) is an extension of first-order logic, and as such has native support for quantification which is needed for generative problem settings.

Ability to provide explanations (C9). Subtask 7 and 8 in Section 4 are inferences that are used to support giving explanations. The implemented configuration system has an implementation of definition 4.14.

6 Related Work

6.1 Other approaches

In different branches of AI research, people have been focusing on configuration software in different settings. The following discussion of knowledge-based approaches is based on a book in recent literature (Felfernig et al. 2014). After the discussion we will compare the ten approaches with our approach (**IDP**).

Historically, the first knowledge-based configuration systems were *rule-based* (**RBS**) (McDermott 1982; Barker and O'Connor 1989). These systems operate on a working memory and if the condition of a rule is fulfilled, it fires and modifies the working memory, applying the conclusion of that rule. Rule-based systems are sensitive to rule orderings. This complicates modification of the rule-base. More importantly, inclusion of problem solving knowledge in the rule-base, makes a rule-base problem specific and focused towards one specific task. This leads to the same problems as in imperative languages. To solve different tasks, more rule-bases have to be built, leading to duplication and fanning out of knowledge, giving issues in maintainability.

Constraint Satisfaction Problems are widely used for tackling configuration problems (Mittal and Frayman 1989; Fleischanderl et al. 1998). A (static[3]) constraint satisfaction problem (**SCSP**) is a triple (V, D, C) of a set of domain variables $V = \{v_1, v_2, \ldots, v_n\}$, a set of domains $\{dom(v_1), dom(v_2), \ldots, dom(v_n)\}$ and set of constraints C. A solution for a SCSP is an assignment S of domain elements $d_i \in dom(v_i)$ to variables v_i, such that each variable has a value in S and constraints C are satisfied by S. A configuration task in SCSP is searching for a solution for a SCSP (V, D, C), where C contains the configuration constraints together with the user preferences. To make efficient CSP configuration systems, different techniques have been used, such as local search (Li et al. 2005), symmetry breaking (Kiziltan et al. 2001) and knowledge compilation techniques such as binary decision diagrams (Hadzic and Andersen 2005). In response to limitations of SCSP in configuration, extensions have been developed. *Dynamic Constraint Satisfaction Problems* (**DCSP**) (Mittal and Falkenhainer 1990) allow for variables to be inactive or irrelevant. If a variable is inactive, it does not need a value in a solution (for example, when configuring a smartphone, no camera resolution is needed if no camera is present). *Generative Constraint Satisfaction Problems* (**GCSP**) (Fleischanderl et al. 1998) extends SCSP with component types and generative constraints.

Janota (2008) studied a mapping of CSP to SAT to use a SAT solver to provide functionality for a configuration system.

There exist many graphical approaches for doing knowledge configuration, and visualizing a configuration model. Kang (1990) used *feature models* (**FM**) for modeling these concepts, while **UML** was proposed in (Falkner and Haselböck 2013). FM and UML configuration approaches have no reasoning algorithms, they need to be used with external algorithms. Karatas et al. (2010) for example combined

[3] In constrast to dynamic and generative constraint satisfaction problem.

feature models with constraint logic programming (CLP) to provide reasoning and automated analysis.

Decidable subsets of first-order logic, *description logics* (**DL**) are used often in context of the semantic web. They have also been used for the development of configuration systems (Hotz et al. 2006; McGuinness and Wright 1998). The trade-off for having decidable subsets of first-order logic is that they are limited in expressivity. This make domain knowledge in these systems less readable, less natural and harder to maintain. An ontology based method was also proposed by Vanden Bossche et al. (2007) using OWL.

Tiihonen et al. developed a configuration system WeCoTin (Tiihonen et al. 2013), based on *Answer Set Programming* (**ASP**). WeCoTin uses Smodels, an ASP system, as inference engine, for propagating consequences of choices. Answer set programming (ASP) is a form of declarative programming based on the stable-model semantics (Gelfond and Lifschitz 1988) for logic programs. The architecture of their reasoning engine is closely related to the reasoning engine we use. Also, in language, many similarities can be identified (Denecker et al. 2012), as they both have their roots in extended logic programming.

Combinations of the above approaches are also proposed in literature, called *hybrid* (**HB**) configuration systems. Typically, they use a DL-based representation for the ontology, together with constraints. They combine reasoning engines from these fields to provide inference (Hotz et al. 2006).

6.2 Comparison of approaches

Felfernig et al. (2014) evaluated all these paradigms with respect to the evaluation criteria from Section 5.2.1. In Table 2, we show this evaluation, together with scores for our implementation in the **IDP** column, based on the discussion of Section 5.2.2.

Table 2. Comparison of systems from Section 6 using criteria from Section 5.2 as in (Felfernig et al. 2014). We use a ✓ to mark good support, a ≈ for partial support and a − to denote that no support is available.

	RBS	SCSP	DCSP	GCSP	SAT	FM	UML	DL	ASP	HB	IDP
C1	-	-	-	-	-	✓	✓	≈	-	≈	-
C2	-	-	-	✓	-	-	✓	✓	✓	✓	✓
C3	-	≈	≈	≈	≈	-	-	≈	≈	≈	≈
C4	≈	-	-	✓	-	-	✓	✓	✓	✓	✓
C5	-	≈	≈	≈	≈	≈	≈	≈	≈	≈	✓
C6	-	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
C7	✓	✓	✓	✓	✓	-	-	≈	≈	≈	≈
C8	≈	-	-	✓	-	-	-	-	≈	✓	✓
C9	≈	✓	✓	✓	≈	-	-	✓	✓	✓	✓

All these approaches are focused towards one specific inference: ontologies are focused on deduction, rule systems are focused on backward/forward chaining, etc. These approaches are less general then the KB paradigm, which is specifically designed to reuse the knowledge for different reasoning tasks. The contributions of

this paper are different from previously discussed approaches: we analyzed IC problems from a Knowledge Representation point of view. This paper is a discussion of possible approaches and the importance of this point of view. We made a study of desired functionalities for an IC system and how we can define logical reasoning tasks to supply these functionalities. As far as we are aware, the language we used in this experiment is more expressive than earlier approaches.

The expressivity of the language is crucial for the usability of the approach. It allows us to address a broader range of applications, moreover it is easier to formalize and maintain the domain knowledge. Not discussed by Felfernig et al. (2014) et al is work by Vlaeminck et al. (2009). They did a preliminary experiment using the KB approach for interactive configuration, also using the FO(·) IDP project. It is on this work that we continue in this paper by analyzing a real-life application of a larger scale and discussing new functionalities and inferences. This theoretical approach benefits from (1) the expressive language to express domain knowledge adequately and (2) the general basic inferences that realise derived inferences in an easy way, supporting the discussed functionalities, resulting in a IC system that scores very well with relation to the evaluation criteria (Table 2).

An interesting remark in Table 2 is that the IDP column resembles the GCSP column, a generalisation of CSP, developed for configuration. The IDP-system has better support for C5 (maintainability), due to the high level modeling language and the strict seperation between domain knowledge and reasoning. GCSP has better efficiency results. This can be partly explained by the fact that CSP uses dedicated algorithms for reasoning over global constraints such as *alldifferent*. The goal of reusing knowledge makes that we typically do not make use of this kind of specific algorithms, since a dedicated algorithm can only be developed with one specific inference in mind.

7 Challenges and Future Work

Interactive configuration problems are part of a broader kind of problems, namely service provisioning problems. Service provisioning is the problem domain of coupling service providers with end users, starting from the request until the delivery of the service. Traditionally, such problems start with designing a configuration system that allows users to communicate their wishes, for which we provided a knowledge-based solution. Once all the information is gathered from a user, it is still necessary to make a plan for the production and delivery of the selected configuration. Hence the configuration problem is followed by a planning problem that shares domain knowledge with the configuration problem but that also has its own domain knowledge about providers of components, production processes, etc. This planning problem then leads to a monitoring problem. Authorizations could be required, payments need to be checked, or it could be that the configuration becomes invalid mid-process. In this case the configuration needs to be redone, but preferably without losing much of the work that is already done. Companies need software that can manage and monitor the whole chain, from initial configuration to final delivery and this without duplication of domain knowledge. This is a problem area

where the KB approach holds great promise but where further research is needed to integrate the KB system with the environment that the company uses to follow up its processes.

Other future work may include language extensions to better support configuration-like tasks. A prime example of this are templates (Dasseville et al. 2015). Oftentimes the theory of a configuration problem contains lots of constraints which are similar in structure. It seems natural to introduce a language construct to abstract away the common parts. Another useful language extension is reification, to talk about the symbols in a specification rather than about their interpretation. Reification allows the system to reason on a meta level about the symbol and for example assign symbols to a category like "Technical" or "Administrative".

8 Conclusion

The KB paradigm, in which a strict separation between knowledge and problem solving is proposed, was analyzed in a class of knowledge intensive problems: interactive configuration problems. As we discussed why solutions for this class are hard to develop, we proposed a novel approach to the configuration problem based on an existing KB system. We analyzed the functional requirements of an IC system and investigated how we can provide these, using logical inferences on a knowledge base. We identified interesting new inference methods and applied them to the interactive configuration domain. We studied this approach in context of a large application, for which we built a proof of concept, using the KB system as an engine, which we extended with the new inferences. As proof of concept, we solved a configuration problem for a large banking company. Results are convincing and open perspectives for further research in service provisioning.

References

Adaptive Planet 2015. Adaptive planet. `http://www.adaptiveplanet.com/`.

Axling, T. and Haridi, S. 1996. A tool for developing interactive configuration applications. *Journal of Logic Programming 26,* 2, 147–168.

Barker, V. E. and O'Connor, D. E. 1989. Expert systems for configuration at digital: XCON and beyond. *Commun. ACM 32,* 3, 298–318.

Bogaerts, B., Jansen, J., Bruynooghe, M., De Cat, B., Vennekens, J., and Denecker, M. 2014. Simulating dynamic systems using linear time calculus theories. *TPLP 14,* 4–5 (7), 477–492.

Bruynooghe, M., Blockeel, H., Bogaerts, B., De Cat, B., De Pooter, S., Jansen, J., Labarre, A., Ramon, J., Denecker, M., and Verwer, S. 2015. Predicate logic as a modeling language: modeling and solving some machine learning and data mining problems with IDP3. *TPLP 15*, 783–817.

Calimeri, F., Ianni, G., and Ricca, F. 2014. The third open answer set programming competition. *TPLP 14,* 1, 117–135.

Dasseville, I., van der Hallen, M., Janssens, G., and Denecker, M. 2015. Semantics of templates in a compositional framework for building logics. *TPLP 15,* 4–5, 681–695.

De Cat, B., Bogaerts, B., Bruynooghe, M., Janssens, G., and Denecker, M. 2016. Predicate logic as a modelling language: The IDP system. *CoRR abs/1401.6312v2.*

Denecker, M., Lierler, Y., Truszczyński, M., and Vennekens, J. 2012. A Tarskian informal semantics for answer set programming. In *ICLP (Technical Communications)*, A. Dovier and V. S. Costa, Eds. LIPIcs, vol. 17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 277–289.

Denecker, M. and Ternovska, E. 2008. A logic of nonmonotone inductive definitions. *ACM Trans. Comput. Log. 9,* 2 (Apr.), 14:1–14:52.

Denecker, M. and Vennekens, J. 2008. Building a knowledge base system for an integration of logic programming and classical logic. In *ICLP*, M. García de la Banda and E. Pontelli, Eds. LNCS, vol. 5366. Springer, 71–76.

Falkner, A. A. and Haselböck, A. 2013. Challenges of knowledge evolution in practice. *AI Communications 26,* 1, 3–14.

Felfernig, A., Hotz, L., Bagley, C., and Tiihonen, J. 2014. *Knowledge-based Configuration: From Research to Business Cases*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Fleischanderl, G., Friedrich, G., Haselböck, A., Schreiner, H., and Stumptner, M. 1998. Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems 13,* 4, 59–68.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *ICLP/SLP*, R. A. Kowalski and K. A. Bowen, Eds. MIT Press, 1070–1080.

Hadzic, T. 2004. A BDD-based approach to interactive configuration. In *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, M. Wallace, Ed. LNCS, vol. 3258. Springer, 797.

Hadzic, T. and Andersen, H. R. 2005. Interactive reconfiguration in power supply restoration. In *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, P. van Beek, Ed. Lecture Notes in Computer Science, vol. 3709. Springer, 767–771.

Heras, F., Morgado, A., and Marques-Silva, J. 2011. Core-guided binary search algorithms for maximum satisfiability. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011.*

Hotz, L., Krebs, T., Deelstra, S., Sinnema, M., and Nijhuis, J. 2006. *Configuration in industrial product families - the ConIPF methodology.* IOS Press, Inc.

Immerman, N. and Vardi, M. Y. 1997. Model checking and transitive-closure logic. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, O. Grumberg, Ed. Lecture Notes in Computer Science, vol. 1254. Springer, 291–302.

Janota, M. 2008. Do SAT solvers make good configurators? In *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops)*, S. Thiel and K. Pohl, Eds. Lero Int. Science Centre, University of Limerick, Ireland, 191–195.

Jansen, J., Dasseville, I., Devriendt, J., and Janssens, G. 2014. Experimental evaluation of a state-of-the-art grounder. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*, O. Chitil, A. King, and O. Danvy, Eds. ACM, 249–258.

Junker, U. 2004. QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, D. L. McGuinness and G. Ferguson, Eds. AAAI Press / The MIT Press, 167–172.

Junker, U. and Mailharro, D. 2003. Preference programming: Advanced problem solving for configuration. *AI EDAM 17,* 1, 13–29.

Kang, K. 1990. *Feature-oriented Domain Analysis (FODA): Feasibility Study ; Technical Report CMU/SEI-90-TR-21 - ESD-90-TR-222.* Software Engineering Inst., Carnegie Mellon Univ.

Karatas, A. S., Oguztüzün, H., and Dogru, A. H. 2010. Mapping extended feature models to constraint logic programming over finite domains. In *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*, J. Bosch and J. Lee, Eds. Lecture Notes in Computer Science, vol. 6287. Springer, 286–299.

Kiziltan, Z., Flener, P., and Hnich, B. 2001. Towards inferring labelling heuristics for CSP application domains. In *KI 2001: Advances in Artificial Intelligence, Joint German/Austrian Conference on AI, Vienna, Austria, September 19-21, 2001, Proceedings*, F. Baader, G. Brewka, and T. Eiter, Eds. LNCS, vol. 2174. Springer, 275–289.

Kleene, S. C. 1952. *Introduction to Metamathematics.* Van Nostrand.

Li, B., Chen, L., Huang, Z., and Zhong, Y. 2005. Product configuration optimization using a multiobjective genetic algorithm. *The International Journal of Advanced Manufacturing Technology 30,* 1, 20–29.

Lynce, I. and Silva, J. P. M. 2004. On computing minimum unsatisfiable cores. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings.*

Marques-Silva, J. and Planes, J. 2008. Algorithms for maximum satisfiability using unsatisfiable cores. In *Design, Automation and Test in Europe, DATE 2008, Munich, Germany, March 10-14, 2008.* 408–413.

McDermott, J. P. 1982. R1: A rule-based configurer of computer systems. *Artif. Intell. 19,* 1, 39–88.

McGuinness, D. L. and Wright, J. R. 1998. An industrial-strength description-logics-based configurator platform. *IEEE Intelligent Systems 13,* 4, 69–77.

Mitchell, D. G. and Ternovska, E. 2005. A framework for representing and solving NP search problems. In *AAAI*, M. M. Veloso and S. Kambhampati, Eds. AAAI Press / The MIT Press, 430–435.

Mittal, S. and Falkenhainer, B. 1990. Dynamic constraint satisfaction problems. In *Proceedings of the 8th National Conference on Artificial Intelligence. Boston, Massachusetts, July 29 - August 3, 1990, 2 Volumes.*, T. Dieterich and W. Swartout, Eds. AAAI/MIT Press, 25–32.

Mittal, S. and Frayman, F. 1989. Towards a generic model of configuraton tasks. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence. Detroit, MI, USA, August 1989*, N. S. Sridharan, Ed. Morgan Kaufmann, 1395–1401.

Pelov, N., Denecker, M., and Bruynooghe, M. 2007. Well-founded and stable semantics of logic programs with aggregates. *TPLP 7,* 3, 301–353.

Piller, F. T., Harzer, T., Ihl, C., and Salvador, F. 2014. Strategic capabilities of mass customization based e-commerce: Construct development and empirical test. In *47th Hawaii International Conference on System Sciences, HICSS 2014, Waikoloa, HI, USA, January 6-9, 2014.* IEEE, 3255–3264.

Pontelli, E. and Son, T. C. 2006. *Justifications* for logic programs under answer set semantics. In *ICLP*, S. Etalle and M. Truszczyński, Eds. LNCS, vol. 4079. Springer, 196–210.

QML 2015. Qml. `http://qmlbook.org/`.

Reiter, R. 1987. A theory of diagnosis from first principles. *Artif. Intell. 32,* 1, 57–95.

Schneeweiss, D. and Hofstedt, P. 2011. Fdconfig: A constraint-based interactive product configurator. In *Applications of Declarative Programming and Knowledge Management - 19th International Conference, INAP 2011, and 25th Workshop on Logic Programming, WLP 2011, Vienna, Austria, September 28-30, 2011, Revised Selected Papers*, H. Tompits, S. Abreu, J. Oetsch, J. Pührer, D. Seipel, M. Umeda, and A. Wolf, Eds. Lecture Notes in Computer Science, vol. 7773. Springer, 239–255.

Shchekotykhin, K. M., Friedrich, G., Rodler, P., and Fleiss, P. 2014. Interactive ontology debugging using direct diagnosis. In *Proceedings of the Third International Workshop on Debugging Ontologies and Ontology Mappings, WoDOOM 2014, co-located with 11th Extended Semantic Web Conference (ESWC 2014), Anissaras/Hersonissou, Greece, May 26, 2014.*, P. Lambrix, G. Qi, M. Horridge, and B. Parsia, Eds. CEUR Workshop Proceedings, vol. 1162. CEUR-WS.org, 39–50.

Shlyakhter, I., Seater, R., Jackson, D., Sridharan, M., and Taghdiri, M. 2003. Debugging overconstrained declarative models using unsatisfiable cores. In *ASE*. IEEE Computer Society, 94–105.

Syrjänen, T. 2006. Debugging inconsistent answer set programs. In *Proceedings of the Eleventh International Workshop on Non-Monotonic Reasoning, NMR 2006, Lake District, UK, 30 May - 1 June*, J. Dix and A. Hunter, Eds. 77–84.

Tiihonen, J., Heiskala, M., Anderson, A., and Soininen, T. 2013. Wecotin - A practical logic-based sales configurator. *AI Commun. 26,* 1, 99–131.

Vanden Bossche, M., Ross, P., MacLarty, I., Van Nuffelen, B., and Pelov, N. 2007. Ontology driven software engineering for real life applications. In *3rd International Workshop on Semantic Web Enabled Software Engineering (SWESE)*.

Vlaeminck, H., Vennekens, J., and Denecker, M. 2009. A logical framework for configuration software. In *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*, A. Porto and F. J. López-Fraguas, Eds. ACM, 141–148.

Wittocx, J., Mariën, M., and Denecker, M. 2008. The idp system: A model expansion system for an extension of classical logic. In *LaSh*, M. Denecker, Ed. ACCO, 153–165.

Wittocx, J., Vlaeminck, H., and Denecker, M. 2009. Debugging for model expansion. In *ICLP*, P. M. Hill and D. S. Warren, Eds. LNCS, vol. 5649. Springer, 296–311.

Zhang, J., Li, S., and Shen, S. 2006. Extracting minimum unsatisfiable cores with a greedy genetic algorithm. In *AI 2006: Advances in Artificial Intelligence, 19th Australian Joint Conference on Artificial Intelligence, Hobart, Australia, December 4-8, 2006, Proceedings*. 847–856.